MATH2071: LAB 6: Solving linear systems

Introduction	Exercise 1
Test matrices	Exercise 2
The linear system "problem"	Exercise 3
Gauß factorization	Exercise 4
Permutation matrices	Exercise 5
PLU factorization	Exercise 6
PLU solution	Exercise 7
A system of ODEs	Exercise 8
Extra Credit: Pivoting	Exercise 9
	Exercise 10
	Exercise 11
	Extra Credit

1 Introduction

In many numerical applications, notably the ordinary and partial differential equations this semester, the single most time-consuming step is the solution of a system of linear equations. We now begin a major topic in Numerical Analysis, *Numerical Linear Algebra*. We assume you are familiar with the concepts of vectors and matrices, matrix multiplication, vector dot products, and the *theory* of the solution of systems of linear equations, the matrix inverse, eigenvalues and so on. We now concentrate on the practical aspects of carrying out the computations required to do linear algebra in a numerical setting.

This lab will take three sessions. If you print this lab, you may prefer to use the pdf version.

We begin by noting several special *test matrices* that we will refer to when trying out algorithms. Then we state the *linear system problem* and consider three methods of solution, using the determinant, the inverse matrix, or Gauß factorization. We will find that factorization into three simply-solved factors is the best way to go, and we will write a m-file to perform the factorization and another to solve systems given the factorization. We will end up with an example using our m-files as part of the numerical solution of a partial differential equation.

2 Test matrices

We will now consider a few simple test matrices that we can use to study the behavior of the algorithms for solution of a linear system. We can use such problems as a quick test that we haven't made a serious programming error, but more importantly, we also use them to study the accuracy of the algorithms. We want test problems which can be made any size, and for which the exact values of the determinant, inverse, or eigenvalues can be determined. Matlab itself offers a "rogues' gallery" of special test matrices through the Matlab function `gallery`, and it offers several special matrices. We will use our own functions for some matrices and Matlab's functions for others.

Some test matrices we will be using are:

- The second difference matrix:
- The Frank matrix,
- The Hilbert matrix.
- The Pascal matrix.

- The magic square matrix.

2.1 The second difference matrix

The second difference matrix arises in approximating the second derivative on equally spaced data. You have seen this matrix and calculated its eigenvalues, eigenvectors, and determinant in Lab 5. The formula is:

$$\mathbf{A}_{i,j} = \begin{cases} 2 & \text{for } j = i \\ -1 & \text{for } |j-i| = 1 \\ 0 & \text{for } |j-i| > 1 \end{cases}$$

So that the matrix looks like:

$$\begin{pmatrix} 2 & -1 & & & & & \\ -1 & 2 & -1 & & & & \\ & -1 & 2 & -1 & & & \\ & & & \ddots & & & \\ & & & -1 & 2 & -1 & \\ & & & & -1 & 2 & \end{pmatrix}$$

where blanks represent zero values. As you have seen, the matrix is positive definite, symmetric, and tridiagonal. The determinant is $(n+1)$ where n denotes the size (A is $n \times n$). Matlab provides a function to generate this matrix called `gallery('tridiag',n)`, but we will use the m-file `dif2.m`.

Do not use the `gallery('tridiag',n)` form in these labs except when instructed to use it. Matlab has two forms of storage for matrices, "sparse" and "full." Everything we have done so far has used the "full" form, but `gallery('tridiag',n)` returns the "sparse" form. The "sparse" matrix form requires some special handling.

Download the m-file, `dif2.m` now.

2.2 The Frank matrix

Row i of the $n \times n$ Frank matrix has the formula:

$$\mathbf{A}_{i,j} = \begin{cases} 0 & \text{for } j < i-2 \\ n+1-i & \text{for } j = i-1 \\ n+1-j & \text{for } j \geq i \end{cases}$$

The Frank matrix for $n = 5$ looks like:

$$\begin{pmatrix} 5 & 4 & 3 & 2 & 1 \\ 4 & 4 & 3 & 2 & 1 \\ 0 & 3 & 3 & 2 & 1 \\ 0 & 0 & 2 & 2 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

The determinant of the Frank matrix is 1, but is difficult to compute accurately because of roundoff errors. This matrix has a special form called *Hessenberg* form wherein all elements below the first subdiagonal are zero. Matlab provides the Frank matrix in its "gallery" of matrices, `gallery('frank',n)`, but we will use an m-file `frank.m`. It turns out that the inverse of the Frank matrix also has a simple formula.

Download the m-file `frank.m` now.

2.3 The Hilbert matrix

The Hilbert matrix is related to the interpolation problem on the interval [0,1]. The matrix is given by the formula $\mathbf{A}_{i,j} = 1/(i+j-1)$. For example, with $n = 5$ the Hilbert matrix is:

$$\begin{pmatrix} \frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{pmatrix}$$

The Hilbert matrix arises in interpolation and approximation contexts because it happens that $\mathbf{A}_{i,j} = \int_0^1 (x^{i-1})(x^{j-1})dx$. The Hilbert matrix is at once nice because its inverse has integer elements and also not nice because it is difficult to compute the inverse accurately using the usual formulæ to invert a matrix.

Matlab has special functions for the Hilbert matrix and its inverse, called `hilb(n)` and `invhilb(n)`, and we will be using these functions in this lab.

2.4 The Pascal matrix

The Pascal matrix is generated by a form of Pascal's triangle. It is defined as $P_{i,j} = \binom{i+j-2}{j-1}$ (the binomial coefficient). For example, with $n = 5$, the Pascal matrix is

$$P = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 6 & 10 & 15 \\ 1 & 4 & 10 & 20 & 35 \\ 1 & 5 & 15 & 35 & 70 \end{pmatrix}$$

As you can tell, the colored portion of this matrix is a Pascal triangle, written on its side.

A related lower triangular matrix can be written (for $n = 5$) as

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 1 & -3 & 3 & -1 & 0 \\ 1 & -4 & 6 & -4 & 1 \end{pmatrix}$$

Except for signs, this matrix is a Pascal triangle written asymmetrically. The matrix

$$L_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 1 & 3 & 3 & 1 & 0 \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix}$$

is truly a Pascal triangle written asymmetrically. Matlab provides a function `pascal` so that `P=pascal(5)`, `L=pascal(5,1)`, and `L1=abs(L)`.

The Pascal matrix, like the Frank matrix, is very ill-conditioned for large values of n. In addition, the Pascal matrix satisfies the conditions

- $\det(P) = \det(L) = \det(L_1) = 1$.
- $L^2 = I$ (I is the identity matrix)
- $P = LL^T = L_1 L_1^T$

More information is available in a Wikipedia article `http://en.wikipedia.org/wiki/Pascal_matrix`

3 The linear system "problem"

The linear system "problem" can be posed in the following way. Find an n-vector $\mathbf{x}$ that satisfies the matrix equation

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

where $\mathbf{A}$ is a $m \times n$ matrix and $\mathbf{b}$ is a m-vector. In Matlab, both $\mathbf{x}$ and $\mathbf{b}$ are column vectors.

You probably already know that there is "usually" a solution if the matrix is square, (that is, if $m = n$). We will concentrate on this case for now. But you may wonder if there is an intelligent response to this problem for the cases of a square but singular matrix, or a rectangular system, whether overdetermined or underdetermined.

At one time or another, you have probably been introduced to several algorithms for producing a solution to the linear system problem, including Cramer's rule (using determinants), constructing the inverse matrix, Gauß-Jordan elimination and Gauß factorization. We will see that it is usually not a good idea to construct the inverse matrix and we will focus on Gauß factorization for solution of linear systems.

We will be concerned about several topics:

Efficiency: what algorithms produce a result with less work?

Accuracy: what algorithms produce an answer that is likely to be more accurate?

Difficulty: what makes a problem difficult or impossible to solve?

Special cases: how do we solve problems that are big? symmetric? banded? singular? rectangular?

4 The inverse matrix

A classic result from linear algebra is the following:
Theorem *The linear system problem (1) is uniquely solvable for arbitrary* $\mathbf{b}$ *if and only if the inverse matrix* $\mathbf{A}^{-1}$ *exists. In this case the solution* $\mathbf{x}$ *can be expressed as* $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

So what's the catch? There are a few:

- Computing the inverse takes a lot of time–more time than is necessary if you only want to know the solution of a particular system.
- In cases when the given matrix actually has very many zero entries, as is the case with the `dif2` matrix, computing the inverse is enormously more difficult and time consuming than merely solving the system. And it is quite often true that the inverse matrix requires far more storage than the matrix itself. The second difference matrix, for example, is an "M-matrix" and it can be proved that all the entries in its inverse are positive numbers. You only need about $3n$ numbers to store the `dif2` matrix because you can often avoid storing the zero entries, but you need n^2 numbers to store its inverse.
- Sometimes the inverse is so inaccurate that it is not worth the trouble to multiply by the inverse to get the solution.

In the following exercises, you will see that constructing the inverse matrix takes time proportional to n^3 (for an $n \times n$ matrix), as does simply solving the system, but that solving the system is several times faster than constructing the inverse matrix. You will also see that matrices with special formats, such as tridiagonal matrices, can be solved very efficiently. And you will see even simple matrices can be difficult to numerically invert.

You will be measuring elapsed time in order to see how long these calculations take. The problem sizes are designed to take a modest amount of time (less than 6 minutes) on newer computers.
WARNING: Most newer computers have more than one processor (core). By default, Matlab will use all the processors available by assigning one "thread" to each available processor. This procedure will mess up our timings, so you should tell Matlab to use only a single thread. To do this, you should start up Matlab with the command line

```
matlab -singleCompThread
```

If you do not know how to start up Matlab from the command line, use the following command at the beginning of your session.

```
maxNumCompThreads(1);
```

Matlab will warn you that this command will disappear in the future, but it should work fine now.

Exercise 1: The Matlab command `inv(A)` computes an approximation for the inverse of a matrix `A`. Do not worry for now just how it does it, but be assured that it uses one of the most efficient and reliable methods available.

Matlab provides the commands `tic` and `toc` to measure computational time. The `tic` command starts the stopwatch, and the `toc` command stops it, either printing the elapsed time or returning its value as in the expression `elapsedTime=toc;`. The times are in seconds. (**Note:** `tic` and `toc` measure *elapsed* time. When a computer is doing more than one thing, the `cputime` function can be used to measure how much computer time is being used for this task alone.)

(a) Copy the following code to a Matlab function m-file named `exer1.m` and modify it to produce information for the table below. Be sure to add comments and your name and the date. Include the file with your summary.

```
function elapsedTime=exer1(n)
% elapsedTime=exer1(n)
% comments

% your name and the date

if mod(n,2)==0
  error('Please use only odd values for n');
end

A = magic(n);    % only odd n yield invertible matrices
b = ones(n,1);   % the right side vector doesn't change the time
tic;
Ainv =  ???      % compute the inverse matrix
xSolution = ???  % compute the solution
elapsedTime=toc;
```

(b) You have seen in lecture that the time required to invert an $n \times n$ matrix should be proportional to n^3. Fill in the following table, where the column entitled "ratio" should contain the ratio of the time for n divided by the time for the preceeding value of n. (**Note:** timings are not precise!

Furthermore, the first time a function is used results in Matlab reading the m-file and "compiling it," and that takes considerable time. The last row of this table may take several minutes!)
Remark: Compute the first line of the table *twice* and use the second value. The first time a function is called involves substantial overhead that later calls do not require.

```
Time to compute inverse matrices
 n     time      ratio
 161  _________
 321  _________  _________
 641  _________  _________
1281  _________  _________
2561  _________  _________
5121  _________  _________
10241 _________  _________
```

(c) Are these solution times roughly proportional to n^3?

Exercise 2: Matlab provides a special operator, the backslash (\) operator, that is designed to solve a linear system without computing the inverse. It is used in the following way, for a matrix `A` and column vector `b`.

```
x=A\b;
```

It may help you to remember this operator if you think of the `A` as being "underneath" the slash. The effect of this operator is to find the solution of the system of equations `A*x=b`.

The backslash looks strange, but there is a method to this madness. You might wonder why you can't just write `x=b/A`. This would put the column vector `b` to the left of the matrix `A` and that is the wrong order of operations.

(a) Copy `exer1.m` to `exer2.m` and replace the inverse matrix computation and solution with an expression using the Matlab "\" command, and fill in the following table

```
Time to compute solutions
 n     time      ratio
 161  _________
 321  _________  _________
 641  _________  _________
1281  _________  _________
2561  _________  _________
5121  _________  _________
10241 _________  _________
```

(b) Are these solution times roughly proportional to n^3?

(c) Compare the times for the inverse and solution and fill in the following table

```
Comparison of times
 n   (time for inverse)/(time for solution)
 161              _________
 321              _________
 641              _________
1281              _________
2561              _________
5121              _________
10241             _________
```

(d) Theory shows that computation of a matrix inverse should take approximately three times as long as computation of the solution. Are your results consistent with theory?

You should be getting the message: **"You should never compute an inverse when all you want to do is solve a system."**
Warning: the "\" symbol in Matlab will work when the matrix `A` is *not square.* In fact, sometimes it will work when `A` actually is a vector. The results are not usually what you expect and no error message is given. Be careful of this potential for error when using "\". A similar warning is true for the `/` (division) symbol because Matlab will try to "interpret" it if you use it with matrices or vectors and you will get "answers" that are not what you intend.

Exercise 3: Some matrices consist of mostly zero entries. These so-called "sparse" matrices sometimes admit extremely efficient methods for solving them. Matlab has a special storage scheme for sparse matrices, and the Matlab function `gallery('tridiag',N)` yields a sparse matrix that is the same as the result from `dif2(N)`.

(a) For N=10 and b=ones(N,1), use the "\" command to solve the system $Ax = b$ once using the matrix from `dif2(N)` and once using the matrix from `gallery('tridiag',N)`. Confirm that the solutions are the same as each other. The easiest way to confirm that two vectors are the same is to compute the relative norm of their difference.

(b) Using the same methodology as in Exercise 2, fill in the following table using the matrix from `gallery('tridiag',N)`.
Note: The matrix sizes in this table increase by a factor of *ten,* not two, because sparse matrix storage allows much larger matrices.

```
  Time to compute solutions
  Size     time      ratio
   10240   --------  -------
  102400   --------  -------
 1024000   --------  -------
10240000
```

(c) Because we are now solving a sparse system, solution time is not necessarily $O(n^3)$. Estimate p so that sparse solution time is $O(n^p)$. Do not forget that the matrix sizes in the table increase by factors of ten, not two.

(d) Compare the time required to solve a system using the full matrix `dif2(10240)` with the time for the sparse matrix computation using `gallery('tridiag',10240)`. You should see a big advantage for sparse storage. One method of sparse storage will be discussed in a later lab.

Exercise 4: Some nonsingular matrices result in systems that are very difficult to solve numerically. You should recall that the condition number of a matrix A, defined as $\|A\|_p\|A^{-1}\|_p$ for $p \geq 1$ (and is equal to the size of the largest eigenvalue over the smallest eigenvalue for symmetric matrices and $p = 2$) is a measure of the likelihood of roundoff errors in solving a matrix equation. In this exercise you will construct a system by picking a solution vector x_{known} and then constructing the right side $b = Ax_{\text{known}}$. Then the known solution can be compared with the solution computed using the "\" command to see how bad a particular solution can be. Consider the following Matlab code:

```
N=10;
A = dif2(N);
xKnown = sqrt( (1:N)' );
b = ???          % compute the right side corresponding to xKnown
xSolution = ???  % compute the solution using "\"
err=norm(xSolution-xKnown)/norm(xKnown)
```

(a) For the matrix defined by `dif2`, fill in the following table. (Recall that the Matlab command "`cond`" yields the condition number and the command "`det`" yields the determinant.)

```
        dif2 matrix
 size    error   determinant cond. no.
   10  _________ ___________  ___________
   40  _________ ___________  ___________
  160  _________ ___________  ___________
```

The `dif2` matrix is about as nicely behaved as a matrix can be. It is neither close to singular nor difficult to invert.

(b) For the matrix defined by `hilb`, fill in the following table.

```
       Hilbert matrix
 size    error   determinant cond. no.
   10  _________ ___________  ___________
   15  _________ ___________  ___________
   20  _________ ___________  ___________
```

The Hilbert matrix is so close to being singular that numerical approximations cannot distinguish it from a singular matrix.

(c) For the matrix defined by `frank`, fill in the following table.

```
         Frank matrix
 size    error   determinant cond. no.
   10  _________ ___________  ___________
   15  _________ ___________  ___________
   20  _________ ___________  ___________
```

The Frank matrix can be proved to have a determinant equal to 1.0 because its eigenvalues come in $(\lambda, 1/\lambda)$ pairs. Thus, it is *not* close to singular. Nonetheless, roundoff errors conspire to make it appear to be singular on a computer. The condition number can alert you to this possibility. You can find some information about the Frank matrix from the Matrix Market, and the references therein.

(d) For the matrix defined by `pascal`, fill in the following table.

```
          Pascal matrix
 size    error   determinant cond. no.
   10  _________ ___________  ___________
   15  _________ ___________  ___________
   20  _________ ___________  ___________
```

The Pascal matrix is another matrix that has a determinant of 1, with eigenvalues occurring in $(\lambda, 1/\lambda)$ pairs. Computing the determinant numerically is subject to even more serious roundoff than for the Frank matrix.

In the following sections, you will write your own routine that does the same task as the Matlab "\" command.

5 Gauß factorization

The standard method for computing the inverse of a matrix is Gaußian elimination. You already know this algorithm, perhaps by another name such as row reduction. You have seen it discussed in class and you can find a discussion on the web at, for example, `http://en.wikipedia.org/wiki/Gaussian_elimination` .

First, we will prefer to think of the process as having two separable parts, the "factorization" or "forward substitution" and "back substitution" steps. The important things to recall include:

1. In the k^{th} step of factorization, one of the remaining equations (rows) is chosen to become the "pivot," and to be associated with x_k. This equation (row) is swapped with the existing k^{th} equation (row).

2. The pivot equation is used to eliminate references to x_k in the remaining equations (*i.e.,* put zeros in position k in the rows below the k^{th} row). After this step, the matrix will have zeros in all positions below and to the left of the diagonal (k, k) position without changing similar zeros generated in earlier steps.

3. After $n-1$ steps of factorization, back substitution begins. The last equation involves a single variable, and can be solved for x_n. But then equation $n-1$ can be solved for x_{n-1}, because x_n is now known, and similarly, the other equations are solved in backwards order.

Different methods for choosing pivots lead to different variants of Gauß factorization. Important ones include:

- The simplest version of Gauß factorization for a matrix A with entries $A_{k,\ell}$ is called *Gauß factorization with no pivoting*, because the pivot for the k^{th} row is simply the diagonal entry $A_{k,k}$. If on step k, the diagonal is zero the method will fail, even though a solution may actually exist. The forward substitution steps begin at the upper left of the matrix and proceed down the diagonal, converting all the entries below the diagonal to zero. The backward substitution steps begin at the lower right and work their way back up the diagonal.

- The method of "Gauß factorization with partial pivoting" chooses as k^{th} pivot the value $A_{\ell,k}$ maximizing $|A_{\ell,k}|$ for $\ell \geq k$. To keep the calculation orderly, this pivot row is actually moved into the k^{th} row of the matrix. In this case, the matrix gradually is transformed into upper triangular shape. For exact arithmetic, if there is a solution, this method is guaranteed to reach it. Moreover, for computer arithmetic, this method has better accuracy properties than no pivoting.

- The method of "Gauß factorization with scaled partial pivoting" chooses as pivot the value $A_{\ell,k}$ maximizing $|A_{\ell,k}|/|A_{\ell,m}|$ for $m \geq \ell$ (*i.e., relative to the rest of row* ℓ) for $\ell \geq k$. This method has better accuracy properties when the matrix is badly scaled, at the expense of more work in choosing the pivot. We will not be looking at this method in this lab.

- The method of "Gauß factorization with complete pivoting" chooses as pivot for the k^{th} row the coefficient $A_{i,j}$ of largest magnitude for $i \geq k$ and $j \geq k$, This method has more bookkeeping than the partial pivoting method, and doesn't produce much improvement, so it is little used.

We now turn to writing code for the Gauß factorization. The discussion in this lab assumes that you are familiar with Gaußian elimination (sometimes called "row reduction"), and is entirely self-contained. Because, however, interchanging rows is a source of confusion, we will assume at first that no row interchanges are necessary.

The idea is that we are going to use the row reduction operations to turn a given matrix **A** into an upper triangular matrix (all of whose elements *below* the main diagonal are zero) **U** and at the same time keep track of the multipliers in the lower triangular matrix **L**. These matrices will be built in such a way that **A** could be reconstructed from the product **LU** at any time during the factorization procedure.

Alert: The fact that $\mathbf{A} = \mathbf{LU}$ even before the computation is completed can be a powerful debugging technique. If you get the wrong answer at the end of a computation, you can test each step of the computation to see where it has gone wrong. Usually you will find an error in the first step, where the error is easiest to understand and fix.

Let's do a simple example first. Consider the matrix

$$\mathbf{A} = \begin{pmatrix} 2 & 4 \\ 1 & 9 \end{pmatrix}.$$

There is only one step to perform in the row reduction process: turn the 1 in the lower left into a 0 by subtracting half the first row from the second. Convince yourself that this process can be written as

$$\begin{pmatrix} 1 & 0 \\ -\frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & 4 \\ 1 & 9 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 0 & 7 \end{pmatrix}. \tag{2}$$

If you want to write, $\mathbf{A} = \mathbf{LU}$, you need to have $\mathbf{L}$ be the inverse of the first matrix on the left in (2). Thus

$$\mathbf{L} = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{pmatrix}$$

and

$$\mathbf{U} = \begin{pmatrix} 2 & 4 \\ 0 & 7 \end{pmatrix}$$

are matrices that describe the row reduction step ($\mathbf{L}$) and its result ($\mathbf{U}$) in such a way that the original matrix $\mathbf{A}$ can be recovered as $\mathbf{A} = \mathbf{LU}$.

Now, suppose you have a 5 by 5 Hilbert matrix (switching to Matlab notation) `A`, and you wish to perform row reduction steps to turn all the entries in the first column below the first row into zero, and keep track of the operations in the matrix `L` and the result in `U`. Convince yourself that the following code does the trick.

```
n=5;
Jcol=1;
A=hilb(5);
L=eye(n);     % square n by n identity matrix
U=A;
for Irow=Jcol+1:n
  % compute Irow multiplier and save in L(Irow,Jcol)
  L(Irow,Jcol)=U(Irow,Jcol)/U(Jcol,Jcol);

  % multiply row "Jcol" by L(Irow,Jcol) and subtract from row "Irow"
  % This vector statement could be replaced with a loop
  U(Irow,Jcol:n)=U(Irow,Jcol:n)-L(Irow,Jcol)*U(Jcol,Jcol:n);
end
```

Exercise 5:

(a) Use the above code to compute the first row reduction steps for the matrix `A=hilb(5)`. Print the resulting matrix `U` and include it in the summary report. Check that all entries in the first column in the second through last rows are zero or roundoff. (If any are non-zero, you have an error somewhere. Find the error before proceeding.)

(b) Multiply `L*U` and confirm for yourself that it equals `A`. An easy way to do this is to compute `norm(L*U-A,'fro')/norm(A,'fro')`. (As you have seen, the Frobenius norm is faster to compute than the default 2-norm, especially for large matrices.)

(c) Use the code given above as a model for an m-file called `gauss_lu.m` that performs Gaußian reduction without pivoting. The function should start out

```
function [L,U]=gauss_lu(A)
% function [L,U]=gauss_lu(A)
```

```
% performs an LU factorization of the matrix A using
% Gaussian reduction.
% A is the matrix to be factored.
% L is a lower triangular factor with 1's on the diagonal
% U is an upper triangular factor.
% A = L * U

% your name and the date
```

(d) Use your routine to find `L` and `U` for the Hilbert matrix of order 5. Include `L` and `U` in your summary.

(e) What is `U(5,5)`, to at least four significant figures?

(f) Verify that `L` is lower triangular (has zeros above the diagonal) and `U` is upper triangular (has zeros below the diagonal).

(g) Confirm that `L*U` recovers the original matrix.

(h) The Matlab expression `R=rand(100,100)` will generate a 100×100 matrix of random entries. *Do not print this matrix–it has 10,000 numbers in it.* Use `gauss_lu` to find its factors `LR` and `UR`.

- Use norms to confirm that `LR*UR=R`, without printing the matrices.
- Use `tril` and `triu` to confirm that `LR` is lower triangular and `UR` is upper triangular, without printing the matrices.

It turns out that omitting pivoting leaves the function you just wrote vulnerable to failure.

Exercise 6:

(a) Compute `L1` and `U1` for the matrix

```
A1 = [ -2   1   0   0   0
        1   0   1  -2   0
        0   0   0   1  -2
        1  -2   1   0   0
        0   1  -2   1   0]
```

The method should fail for this matrix, producing matrices with infinity (`inf`) and "Not a Number" (`NaN`) in them.

(b) On which step of the decomposition (values of `Irow` and `Jcol`) does the method fail? Why does it fail?

(c) What is the determinant of `A1`? What is the condition number (`cond(A1)`)? Is the matrix singular or ill-conditioned?

In Gaußian elimination it is entirely possible to end up with a zero on the diagonal, as the previous exercise demonstrates. Since you cannot divide by zero, the procedure fails. It is also possible to end up with small numbers on the diagonal and big numbers off the diagonal. In this case, the algorithm doesn't fail, but roundoff errors can overwhelm the procedure and result in wrong answers. One solution to these difficulties is to switch rows and columns around so that the largest remaining entry in the matrix ends up on the diagonal. Instead of this "full pivoting" strategy, it is almost as effective to switch the rows only, instead of both rows and columns. This is called "partial pivoting." In the following section, permutation matrices are introduced and in the subsequent section they are used to express Gaußian elimination with partial pivoting as a "PLU" factorization.

6 Permutation matrices

In general, a matrix that looks like the identity matrix but with two rows interchanged is a matrix that causes the interchange of those two rows when multiplied by another matrix. A permutation matrix can actually be a little more complicated than that, but all permutation matrices are products of matrices with a single interchanged pair of rows.

Exercise 7: Consider the two permutation matrices

```
P1=[1 0 0 0
    0 0 1 0
    0 1 0 0
    0 0 0 1];

P2=[1 0 0 0
    0 0 0 1
    0 0 1 0
    0 1 0 0];
```

(a) Compute the product of `P1` with the magic square matrix of order 4, `A=magic(4)`, `A2=P1*A`. If the result is not the matrix `A` with rows two and three interchanged, then you made an error. Please include the matrix `A2` in your summary.

(b) Multiplying on the right by a permutation matrix permutes the *columns* instead of the rows. Compute the product of the magic square matrix of order 4, `A=magic(4)`, with `P1`, `A3=A*P1`. If the result is not the matrix `A` with columns two and three interchanged, then you made an error. Please include the matrix `A3` in your summary.

(c) What permutation matrix would interchange the first and fourth rows of A?

(d) Compute the product `P=P1*P2`. The product of two permutation matrices is again a permutation matrix, so `P` is also a permutation matrix, but it is slightly more complicated than before. You should observe that, despite the fact that `P1` and `P2` are symmetric permutation matrices, the permutation matrix `P` is *not* symmetric. Please include the matrix `P` in your summary.

(e) Compute the product `A4=P*A` and notice that it is the original `A` with mixed-up rows. In general, a permutation matrix is one in which each row and each column has exactly one 1 in it and all other entries are zero. Please include the matrix `A4` with your summary.

7 PLU factorization

You already have written `gauss_lu.m` to do Gauß eliminiation without pivoting. Now, you are going to add pivoting to the process. The previous section should have convinced you that pivoting is the same thing as multiplying by a permutation matrix. Just as in `gauss_lu.m`, where you started out with `L` equal to the identity matrix and saved the row operations in it, you will start off with a permutation matrix `P=eye(n)` and save the row interchanges in it.

Exercise 8:

(a) Copy your file `gauss_lu.m` to a file called `gauss_plu.m`. Change the signature line to

```
function [P,L,U]=gauss_plu(A)
```

and change the comments to describe the new output matrix `P`.

(b) Add the initialization statement `P = eye(n)` near the beginning.

(c) Add the following code to the beginning of the loop on `Jcol`, (*i.e.,* just before the `for Irow= ...` statement. This code will be performed before any other processing inside that loop.

```
% First, choose a pivot row by finding the largest entry
% in column=Jcol on or below position k=Jcol.
% The row containing the largest entry will then be switched
% with the current row=Jcol
[ colMax, pivotShifted ] = max ( abs ( U(Jcol:n, Jcol ) ) );

% The value of pivotShifted from max needs to be adjusted.
% See help max for more information.
pivotRow = Jcol+pivotShifted-1;

if pivotRow ~= Jcol     % no pivoting if pivotRow==Jcol
  U([Jcol, pivotRow], :)       = U([pivotRow, Jcol], :);
  L([Jcol, pivotRow], 1:Jcol-1)= L([pivotRow, Jcol], 1:Jcol-1);
  P(:,[Jcol, pivotRow])        = P(:,[pivotRow, Jcol]);
end
```

The reason that `pivotShifted` must be adjusted to get `pivotRow` is because, for example, if `J(jcol+1,jcol)` happens to be the largest entry in the column, then the `max` function will return `pivotShifted=2`, not `pivotShifted=(jcol+1)`. This is because `(jcol+1)` is the second entry in the vector `(Jcol:n)`.

(d) **Temporarily** add the following statement at the end of the loop on `Jcol` (after the loop on `Irow`)

```
% TEMPORARY DEBUGGING CHECK
if norm(P*L*U-A,'fro')> 1.e-12 * norm(A,'fro')
  error('If you see this message, there is a bug!')
end
```

(e) Test `gauss_plu.m` on the matrix `A=pascal(5)`, and check that `P*L*U=A`. You should observe that `P` is *not* the identity matrix. Please include `P`, `L`, and `U` in your summary.

(f) Test `gauss_plu.m` on the matrix `A1` from Exercise 6 above, and check that `P1*L1*U1=A1`. Recall that this was the matrix for which `gauss_lu.m` failed. You should not get any error messages and you should not have NaN or inf in the factors. Please include the matrices `P1`, `L1` and `U1` with your summary.

(g) Remove the "`TEMPORARY DEBUGGING CHECK`" that you inserted above. This check involves too much computation and will mess up the timings in later exercises.

8 PLU solution

We have seen how to factor a matrix $\mathbf{A}$ into the form:

$$\mathbf{A} = \mathbf{PLU}$$

where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is a unit lower triangular matrix, and $\mathbf{U}$ is an upper triangular matrix. To solve a matrix equation involving $\mathbf{A}$, we use this factor information to "peel away" the multiplication a

step at a time:

$$\begin{aligned}
\mathbf{Ax} &= \mathbf{b} \\
\mathbf{PLUx} &= \mathbf{b} \\
\mathbf{LUx} &= \mathbf{P}^{-1}\mathbf{b} \\
\mathbf{Ux} &= \mathbf{L}^{-1}(\mathbf{P}^{-1}\mathbf{b}) \\
\mathbf{x} &= \mathbf{U}^{-1}(\mathbf{L}^{-1}\mathbf{P}^{-1}\mathbf{b})
\end{aligned}$$

However, instead of explicitly computing the inverses of the matrices, we use special facts about their form. It's not the inverse matrix itself that we want, but the inverse times the right hand side.

Let's consider the factored linear system once more. It helps to make up some names for variables and look at the problem in a different way:

$$\begin{aligned}
\mathbf{P}(\mathbf{LUx}) &= \mathbf{b} \\
\mathbf{P}(\mathbf{z}) &= \mathbf{b} \\
\mathbf{L}(\mathbf{Ux}) &= \mathbf{z} \\
\mathbf{L}(\mathbf{y}) &= \mathbf{z} \\
\mathbf{Ux} &= \mathbf{y} \\
\mathbf{Ux} &= \mathbf{y} \\
\mathbf{x} &= \mathbf{U}^{-1}\mathbf{y} \\
\mathbf{x} &= \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}^{-1}\mathbf{b}
\end{aligned}$$

Notice that all I have done is to use parentheses to group factors, and name them. But you should now be able to see what an algorithm for solving this problem might look like.

In summary, the **PLU Solution Algorithm** is: To solve $\mathbf{Ax} = \mathbf{b}$ after finding the factors $\mathbf{P}$, $\mathbf{L}$, $\mathbf{U}$ and the right hand side $\mathbf{b}$,

1. Solve $\mathbf{Pz} = \mathbf{b}$;
2. Solve $\mathbf{Ly} = \mathbf{z}$; and,
3. Solve $\mathbf{Ux} = \mathbf{y}$.

Ouch! I promised a simple algorithm, but now instead of having to solve one system, we have to solve three, and we have three different solution vectors running around. But things actually have gotten better, because these are really simple systems to solve:

1. The solution of $\mathbf{Pz} = \mathbf{b}$ is $\mathbf{z} = \mathbf{P}^T\mathbf{b}$;
2. It is easy to solve $\mathbf{Ly} = \mathbf{z}$ because $\mathbf{L}$ is lower triangular and you can start at the top left; and
3. It is easy to solve $\mathbf{Ux} = \mathbf{y}$ because $\mathbf{U}$ is upper triangular and you can start at the bottom right.

The next exercise illustrates this process on a simple matrix and the following exercise has you write a Matlab routine to carry out the process in general. It also provides a worked-out example you can use to check Matlab code.

Exercise 9: Consider the simple matrix factorization

$$\begin{aligned}
\mathbf{A} &= \mathbf{PLU} \\
\begin{pmatrix} 2 & 6 & 12 \\ 1 & 3 & 8 \\ 4 & 4 & 8 \end{pmatrix} &= \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.25 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} 4 & 4 & 8 \\ 0 & 4 & 8 \\ 0 & 0 & 2 \end{pmatrix}
\end{aligned} \tag{3}$$

(a) Using the following matrices,

```
P = [0     1     0
     0     0     1
     1     0     0];

L = [1      0    0
     0.5    1    0
     0.25   0.5  1];

U = [4     4     8
     0     4     8
     0     0     2];
```

confirm that $\mathbf{A} = \mathbf{PLU}$.

(b) Using pencil and paper, go through the steps of solving the system, with right hand side `b=[28; 18; 16]` (note the semicolon to make `b` a column vector), using the PLU factorization. This will serve as a test problem for the code.

```
Step
  0    b = [          28;          18 ;          16 ]
  1    z = [  __________; __________ ; __________ ]
  2    y = [  __________; __________ ; __________ ]
  3    x = [  __________; __________ ; __________ ]
```

In the following exercise, you will be writing a routine `plu_solve.m`, to solve the linear system (1) by solving the permutation, lower and upper triangular systems. Because this is a three stage process, we will divide it into four separate m-files: `u_solve.m`, `l_solve.m`, `p_solve.m`, and `plu_solve.m`. The `plu_solve` routine will look like:

```
function x = plu_solve ( P, L, U, b )
% function x = plu_solve ( P, L, U, b )
% solve a factored system
% P=permutation matrix
% L=lower-triangular
% U=upper-triangular
% b=right side
% x=solution

% your name and the date

% Solve P*z=b for z using p_solve
z = p_solve(P,b);

% Solve L*y=z for y using l_solve
y = l_solve(L,z);

% Solve U*x=y for x using u_solve
x = u_solve(U,y);
```

Exercise 10:

(a) The easiest of these routines by far is `p_solve` because the matrix $\mathbf{P}$ is orthogonal so its solution is $\mathbf{z} = \mathbf{P}^T\mathbf{b}$. This is a one-liner in Matlab. Replace the question marks with code in the following outline.

```
function z=p_solve(P,b)
% z=p_solve(P,b)
% P is an orthogonal matrix
% b is the right hand side
% z is the solution of P*z=b

% your name and the date

z= ???
```

(b) Test your routine using the $\mathbf{P}$ matrix and $\mathbf{b}$ vector from the previous exercise. Do not continue until you are confident your file is correct.

(c) The second routine is still not so hard because all you have to do is start at the top and run down the rows. Replace the question marks with code in the following outline.

```
function y=l_solve(L,z)
% y=l_solve(L,z)
% L is a lower-triangular matrix whose diagonal entries are 1
% z is the right hand side
% y is the solution of L*y=z

% your name and the date

% set n for convenience and simplicity
n=numel(z);
% initialize y to zero and make sure it is a column vector
y=zeros(n,1);

% first row is really an easy one, especially since the diagonal
% entries of L are equal to 1
Irow=1;
y(Irow)=z(Irow);

for Irow=2:n
  rhs = z(Irow);
  for Jcol=1:Irow-1
    rhs = rhs - ???
  end
  y(Irow) = ???
end
```

(d) Test this routine using the $\mathbf{L}$ matrix and $\mathbf{z}$ vector from the previous exercise. Do not continue until you are confident your file is correct.

(e) The third routine is a little harder because you have to start at the *bottom* and run *up* the rows. Replace the question marks with code in the following outline.

```
function x = u_solve(U,y)
% x=u_solve(U,y)
% U is an upper-triangular matrix
```

```
% y is the right hand side
% x is the solution of U*x=y

% your name and the date

% set n for convenience and simplicity
n=numel(y);
% initialize y to zero and make sure it is a column vector
x=zeros(n,1);

% last row is the easy one
Irow=n;
x(Irow)=y(Irow)/U(Irow,Irow);

% the -1 in the middle means it is going up from the bottom
for Irow=n-1:-1:1
  rhs = y(Irow);
  % the following loop could also be written as a single
  % vector statement.
  for Jcol=Irow+1:n
    rhs = rhs - ???
  end
  x(Irow) = ???
end
```

(f) Test this routine using the **U** matrix and **y** vector from previous exercise. Do not continue until you are confident your file is correct.

(g) Now put them all together and verify your solution by calculating

```
x = plu_solve(P,L,U,b)
```

for the matrices and right side vector in the previous exercise. Double-check your work by showing

```
relErr = norm(P*L*U*x-b)/norm(b)
```

is zero or roundoff.

(h) As a final test, solve a large system with randomly generated coefficients and right side. What is `relErr`?

```
A = rand(100,100);
x = rand(100,1);
b = A*x;
```

and the rest of the code as in the previous part of this Exercise. The error `relErr` will usually be a roundoff sized number. There is always some chance that the randomly generated matrix `A` will end up poorly conditioned or singular, though. If that happens, generate `A` and `x` again: they will be different this time.

You might wonder why we don't combine the m-files `gauss_plu.m` and `plu_solve.m` into a single routine. One reason is that there are other things you might want to do with the factorization instead of solving a system. For example, one of the best ways of computing a determinant is to factor the matrix. (The determinant of any permutation matrix is ± 1, the determinant of `L` is 1 because `L` has all diagonal entries equal to 1, and the determinant of `U` is the product of its diagonal entries.) Another reason is that the factorization is much more expensive (*i.e.,* time-consuming) than the solve is, and you might be able to solve the system many times using the same factorization. This latter case is presented in the following section.

9 A system of ODEs

You saw how to numerically solve certain ordinary differential equations in Labs 1-4. You saw that you could solve a scalar linear ODE using the backward Euler method but for nonlinear and for systems of equations, you used Newton's method to solve the timestep equations. It turns out that *linear systems* can be solved using matrix solution methods without resort to Newton's method. At each time step in that solution, you need to solve a linear system of equations, and you will use `plu_factor` and `plu_solve` for this task. You will also see the advantage of splitting the solution into a "factor" step and a "solve" step.

Consider a system of ordinary differential equations.

$$\frac{d\mathbf{u}}{dt} = \mathbf{A}\mathbf{u} \tag{4}$$

where $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{x}$ is an n-vector. Since $\mathbf{u}$ is a vector, it has subscripts, but since we are solving an ODE numerically, there will be time steps as well. We will write time levels using a superscript here so that it is a little clearer. Applying the implicit Euler method to (4) yields the system of equations

$$\begin{aligned} (\mathbf{I} - \Delta t\mathbf{A})\mathbf{u}^{k+1} &= \mathbf{u}^k \\ \mathbf{u}^{k+1} &= (\mathbf{I} - \Delta t\mathbf{A})^{-1}\mathbf{u}^k \end{aligned} \tag{5}$$

where $\mathbf{I}$ denotes the identity matrix.

In Matlab, we will represent the vector $\mathbf{u}^k$ as `u(:,k)`.

The following code embodies the procedure described above. Copy it into a *script* m-file called `bels.m` (for Backward Euler Linear System). This particular system, using the negative of the `dif2` matrix, turns out to be a discretization of the time-dependent heat equation. The variable $\mathbf{u}$ represents temperature and is a function of time and also of a one-dimensional spatial variable $x_j = j\,dx$ for fixed dx. The plot is a compact way to illustrate the vector $\mathbf{u}$ with its subscript measured along the horizontal axis. The red line is the initial shape, the green line is what the final limiting shape would be if integration were continued to $t = \infty$, and the blue lines are the shapes at intermediate times.

```
% ntimes = number of temporal steps from t=0 to t=1 .
ntimes=30;
% dt = time increment
dt=1/ntimes;
t(1,1)=0;

% N is the dimension of the space
N=402;

% initial values
u(1:N/3  ,1) = linspace(0,1,N/3)';
u(N/3+1:2*N/3,1) = linspace(1,-1,N/3)';
u(2*N/3+1:N,1) = linspace(-1,0,N/3)';

% discretization matrix is -dif2 multiplied
% by a constant to make the solution interesting
A= -N^2/5 * dif2(N);
EulerMatrix=eye(N)-dt*A;

tic;
for k = 1 : ntimes
  t(1,k+1) = t(1,k) + dt;
  [P,L,U] = gauss_plu(EulerMatrix);
```

```
  u(:,k+1) = plu_solve(P,L,U,u(:,k));
end
CalculationTime=toc

% plot the solution at sequence of times
plot(u(:,1),'r')
hold on
for k=2:3:ntimes
  plot(u(:,k))
end
plot(zeros(size(u(:,1))),'g')
title 'Solution at selected times'
hold off
```

Exercise 11:

(a) Double-check to be sure the "debugging check" has been eliminated from `gauss_plu.m`.

(b) Use cut-and-paste to copy the above code to a script m-file named `bels.m`. Record the time this solution takes. Save your solution for comparison by copying it (`usave=u`).

(c) Next, change `bels.m` to `bels1.m` so that it uses the routine `gauss_plu.m` *outside the loop* and is executed *only once* to factor the matrix `EulerMatrix` into `P, L`, and `U`. Then use `plu_solve.m` inside the loop to solve the system. Check that, for the case `ntimes=1`, `bels.m` and `bels1.m` yield the same solution in about the same elapsed time.

(d) Use `bels1.m` with `ntimes=30` and record the time. The time spent for this case should be much smaller than for `bels.m` because factorization takes a lot more arithmetic than solution.

(e) Compare the two solutions by computing the norm of the difference (`norm(usave-u,'fro')`) to show they agree. Include your `bels1.m` with your summary.

You should be aware that the system matrix in this exercise is sparse, but we have not taken advantage of that fact here. Neither have we taken advantage of the fact that the matrix is symmetric and positive definite (so that pivoting is unnecessary). As you have seen, a lot of time could be saved by using `gallery(tridiag,n)` and sparse forms of `P`, `L`, and `U`.

If you happened to try Exercise 11 using the "\" form, you would find that it is substantially faster than our routines. It is faster because it takes advantage of programming efficiencies that are beyond the scope of the course, not because it uses different methods. One such efficiency would be the replacement of as many loops as possible with vector statements.

10 Extra credit: Pivoting (8 points)

In Exercise 8 above, you added pivoting code to `gauss_plu.m`. That code is reproduced below

```
  [ colMax, pivotShifted ] = max ( abs ( U(Jcol:n, Jcol ) ) );

  pivotRow = Jcol+pivotShifted-1;

  if pivotRow ~= Jcol     % no pivoting if pivotRow==Jcol
    U([Jcol, pivotRow], :)       = U([pivotRow, Jcol], :);
    L([Jcol, pivotRow], 1:Jcol-1)= L([pivotRow, Jcol], 1:Jcol-1);
    P(:,[Jcol, pivotRow])        = P(:,[pivotRow, Jcol]);
  end
```

In this section, you will write your own code to accomplish the same end.
Note: Given a projection matrix P that represents a single interchange of two rows, so that PB results in two rows of a matrix B being interchanged, then the product BP^T represents the interchange of two columns.

The key to the proof of correctness of the Gauß factorization strategy is the fact that, at each step, the equality $A = PLU$ holds, where

1. P is a projection matrix that represents the product of row interchange operations,
2. L is a lower triangular matrix with ones on the diagonal that represents the product of "row reduction operations" (one row minus a multiple of a previous row), and
3. U is partially upper triangular, with zeros below the diagonal in the first several columns, and each step of the algorithm results in one more column with zeros below the diagonal.

Clearly, when the algorithm is complete, U is (fully) upper trianular.

Suppose you have completed several steps of the Gauß factorization, and on the subsequent step have discovered that pivoting is necessary. Call the pivoting matrix Π. Recall that $\Pi^T\Pi = \Pi\,\Pi^T = I$ so that you can write

$$A = PLU = P\Pi^T\Pi\,L\,\Pi^T\Pi\,U = \overline{P}\,\overline{L}\,\overline{U}.$$

The matrix $\overline{P} = P\Pi^T$ clearly satisfies Condition 1 above, and the matrix $\overline{U} = \Pi U$ satisfies Condition 3. For Condition 2, note that L can be written as

$$L = (I + \Lambda)$$

where Λ is lower triangular, has zeros on the diagonal, and is nonzero only for its first several columns. As a consequence, $\Lambda\Pi^T = \Lambda$ (see the exercise below). Hence

$$\overline{L} = \Pi\,L\,\Pi^T = (\Pi(I + \Lambda)\Pi^T = I + \Pi\Lambda,$$

thus confirming Condition 2.

Exercise 12:

(a) Consider the following matrix

```
A = [ 6   1   1   0   1   1
      1   6   1   1   0   1
      1   1   6   1   1   0
      1   1   0   1   1   6
      1   0   1   1   6   1
      0   1   1   6   1   1 ];
```

Use your `gauss_plu` function on this matrix and confirm that only a single interchange of rows four and six is the result of the pivoting strategy.

(b) Make a copy of `gauss_plu.m` called `new_plu.m`. Change it so that it performs only three reduction steps, to the point just before the pivoting is to take place. Please include the matrices `P`, `L`, and `U` in your summary file.

(c) Construct the matrix Π describing the next row interchange (interchange rows 4 and 6) and construct the matrix $\Lambda = L - I$. Show that $\Lambda\Pi^T = \Lambda$. Explain, in no more than one sentence, why this is an example of a generally true relation.

(d) Restore `new_plu.m` so it does its reduction on all rows, not just the first three. Rewrite the pivoting code in it using loops, without using colons, and using the `max` function only for scalar variables. Carefully test your code to be sure results from `new_plu.m` agree with those from `gauss_plu.m`. Include a description of your testing in your summary. Your testing strategy will be part of your grade.

Last change `$Date: 2017/01/29 22:18:18 $`